10

Errors and Exceptions

WHAT'S IN THIS CHAPTER?

- Looking at the exception classes
- Using `try...catch...finally` to capture exceptions
- Filtering exceptions
- Creating user-defined exceptions
- Retrieving caller information

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/ErrorsAndExceptions`.

The code for this chapter is divided into the following major examples:

- SimpleExceptions
- ExceptionFilters
- RethrowExceptions
- SolicitColdCall
- CallerInformation

All the sample projects have nullable reference types configured.

HANDLING ERRORS

Errors happen, and they are not always caused by the person who coded the application. Sometimes your application generates an error because of an action that was initiated by the end user of the application, or it might be simply due to the environmental context in which your code is running. In any case, you should anticipate errors occurring in your applications and code accordingly.

.NET has enhanced the ways in which you deal with errors. C#'s mechanism for handling error conditions enables you to provide custom handling for each type of error condition and to separate the code that identifies errors from the code that handles them.

Your programs should be capable of handling any possible errors that might occur. For example, in the middle of some complex processing of your code, you might discover that it doesn't have permission to read a file; or, while it is sending network requests, the network might go down. In such exceptional situations, it is not enough for a method to simply return an appropriate error code—there might be 15 or 20 nested method calls, so what you really want the program to do is jump back up through all those calls to exit the task completely and take the appropriate counteractions. The C# language has very good facilities for handling this kind of situation through the mechanism known as *exception handling*.

This chapter covers catching and throwing exceptions in many different scenarios. You see exception types from different namespaces and their hierarchy, and you find out how to create custom exception types. You discover different ways to catch exceptions—for example, how to catch exceptions with the exact exception type or a base class. You also see how to deal with nested `try` blocks and how you could catch exceptions that way. For code that should be invoked no matter whether an exception occurs or the code continues with any error, you are introduced to creating `try/finally` code blocks.

By the end of this chapter, you will have a good grasp of advanced exception handling in your C# applications.

PREDEFINED EXCEPTION CLASSES

In C#, an exception is an object created (or *thrown*) when a particular exceptional error condition occurs. This object contains information that should help identify the problem. Although you can create your own exception classes (and you do so later), .NET includes many predefined exception classes. You can find all the .NET exceptions in the Microsoft documentation in the list of classes that derive from the `Exception` base class at `https://docs.microsoft.com/dotnet/api/system.exception`. This large list only shows the exceptions that directly derive from `Exception`. When you click every other base class, for example `https://docs.microsoft.com/en-us/dotnet/api/system.systemexception`, you see another large list of exception classes driving from `SystemException`.

Here some really important exception types are explained:

- When receiving arguments with a method, you should check the arguments to determine whether they contain values as expected. If this is not the case, you can throw an `ArgumentException` or an exception that derives from this exception class like `ArgumentNullException` and `ArgumentOutOfRangeException`.
- The `NotSupportedException` is thrown when a method is not supported—for example, from classes that implement interfaces but do not implement all the members of the interface. You should not invoke methods that throw this exception, so you should not handle this exception. This exception gives good information during development time to change your code.
- The `StackOverflowException` is thrown by the runtime when the area of memory allocated for the stack is full. A stack overflow can occur if a method continuously calls itself recursively. This is generally a fatal error because it prevents your application from doing anything apart from terminating (in which case it is unlikely that even the finally block will execute). Trying to handle errors like this yourself is usually pointless; instead, you should have the application gracefully exit.

- The `OverflowException` is thrown after an arithmetic calculation and the value does not fit into the variable type—in a checked context. Remember, you can create checked contexts with the checked keyword. Within the checked context, if you attempt to cast an `int` containing a value of –40 to an `uint`, an `OverflowException` is thrown. Creating checked contexts is covered in Chapter 5, "Operators and Casts."
- For exceptions with file I/O, the base class `IOException` is defined. `FileLoadException`, `FileNotFoundException`, `EndOfStreamException`, and `DriveNotFoundException` are some examples that derive from this base class.
- The `InvalidOperationException` is typically thrown if methods of a class are not invoked in the correct order, for example, an initialization call was missing.
- The `TaskCanceledException` is thrown on cancellation of tasks or timeouts.

> **NOTE** *Read the Microsoft documentation for methods you invoke. Every method that might throw exceptions has documentation in the section "Exceptions" of which exceptions can be thrown. For example, the documentation for* `GetStreamSync` *from the* `HttpClient` *class (*`https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient.getstreamasync`*) lists* `ArgumentNullException`, `HttpRequestException`, *and* `TaskCanceledException`.

> **NOTE** *Looking at the hierarchy of the exception types, you might be wondering about the purpose of the base classes* `SystemException` *and* `ApplicationException`. *With the original design of .NET, it was planned to have* `SystemException` *as a base class for all exceptions thrown from the runtime, and* `ApplicationException` *as a base class for all application-defined exceptions. As it turned out,* `ApplicationException` *was rarely used as the base class for specific exceptions. Nowadays, it's okay to directly derive your custom exception from the* `Exception` *base class.*

CATCHING EXCEPTIONS

Given that .NET includes a selection of predefined base class exception objects, this section describes how you use them in your code to trap error conditions. In dealing with possible error conditions in C# code, you typically divide the relevant part of your program into blocks of three different types:

- `try` blocks encapsulate the code that forms part of the normal operation of your program and that might encounter some serious error conditions.
- `catch` blocks encapsulate the code dealing with the various error conditions that your code might have encountered by working through any of the code in the accompanying `try` block. This block could also be used for logging errors.
- `finally` blocks encapsulate the code that cleans up any resources or takes any other action that you normally want handled at the end of a `try` or `catch` block. It is important to understand that the `finally` block is executed whether an exception is thrown. Because the purpose of the `finally` block is to contain cleanup code that should always be executed, the compiler flags an error if you place a

`return` statement inside a `finally` block. An example of using the `finally` block is closing any connections that were opened in the `try` block. Understand that the `finally` block is completely optional. If your application does not require any cleanup code (such as disposing of or closing any open objects), then there is no need for this block.

> **NOTE** *`finally` blocks are a great way to write some cleanup code. This block is executed in every case the `try`/`finally` block is left. It's executed on a successful return of the `try` block and also executed if an exception is thrown.*

The following steps outline how these blocks work together to trap error conditions:

1. The execution flow first enters the `try` block.
2. If no errors occur in the `try` block, execution proceeds normally through the block, and when the end of the `try` block is reached, the flow of execution jumps to the `finally` block if one is present (step 5). However, if an error does occur within the `try` block, execution jumps to a `catch` block (step 3).
3. The error condition is handled in the `catch` block.
4. At the end of the `catch` block, execution automatically transfers to the `finally` block if one is present.
5. The `finally` block is executed (if present).

The C# syntax used to bring all this about looks roughly like this:

```
try
{
  // code for normal execution
}
catch
{
  // error handling
}
finally
{
  // clean up
}
```

A few variations on this theme exist:

- You can omit the `finally` block because it is optional.
- You can also supply as many `catch` blocks as you want to handle specific types of errors. However, you don't want to get too carried away and have a huge number of `catch` blocks.
- You can define filters with catch blocks to catch the exception with the specific block only if the filter matches.
- You can omit the `catch` blocks altogether, in which case the syntax serves not to identify exceptions but to guarantee that code in the `finally` block will be executed when execution leaves the `try` block. This is useful if the `try` block contains several exit points. If you don't write a `catch` block, a `finally` block is required. You cannot use just a `try` block. What would a `try` block be good for without `try` or `finally`?

So far so good, but the question that has yet to be answered is this: if the code is running in the `try` block, how does it know when to switch to the `catch` block if an error occurs? If an error is detected, the code does something known as *throwing an exception*. In other words, it instantiates an exception object class and throws it:

```
throw new OverflowException();
```

Here, you have instantiated an exception object of the `OverflowException` class. As soon as the application encounters a `throw` statement inside a `try` block, it immediately looks for the `catch` block associated with that `try` block. If more than one `catch` block is associated with the `try` block, it identifies the correct `catch` block by checking which exception class the `catch` block is associated with. For example, when the `OverflowException` object is thrown, execution jumps to the following `catch` block:

```
catch (OverflowException ex)
{
  // exception handling here
}
```

In other words, the application looks for the `catch` block that indicates a matching exception class instance of the same class (or of a base class).

With this extra information, you can extend the `try` block with multiple `catch` blocks. Assume, for the sake of argument, that two possible serious errors can occur in the `try` block: an overflow and an array out of bounds. Assume also that your code contains two Boolean variables, `Overflow` and `OutOfBounds`, which indicate whether these conditions exist. You have already seen that a predefined exception class exists to indicate overflow (`OverflowException`); similarly, an `IndexOutOfRangeException` class exists to handle an array that is out of bounds.

Now your `try` block looks like this:

```
try
{
  // code for normal execution
  if (Overflow == true)
  {
    throw new OverflowException();
  }
  // more processing
  if (OutOfBounds == true)
  {
    throw new IndexOutOfRangeException();
  }
  // otherwise continue normal execution
}
catch (OverflowException ex)
{
  // error handling for the overflow error condition
}
catch (IndexOutOfRangeException ex)
{
  // error handling for the index out of range error condition
}
finally
{
  // clean up
}
```

This is because you can have `throw` statements that are nested in several method calls inside the `try` block, but the same `try` block continues to apply even as execution flow enters these other methods. If the application

encounters a `throw` statement, it immediately goes back up through all the method calls on the stack, looking for the end of the containing `try` block and the start of the appropriate `catch` block. During this process, all the local variables in the intermediate method calls will correctly go out of scope. This makes the `try...catch` architecture well suited to the situation described at the beginning of this section, whereby the error occurs inside a method call that is nested inside 15 or 20 method calls, and processing must stop immediately.

As you can probably gather from this discussion, `try` blocks can play a significant role in controlling the flow of your code's execution. However, it is important to understand that exceptions are intended for exceptional conditions, hence their name. You wouldn't want to use them as a way of controlling when to exit a `do...while` loop.

Exceptions and Performance

Exception handling has a performance implication. In cases that are common, you shouldn't use exceptions to deal with errors. For example, when converting a string to a number, you can use the `Parse` method of the `int` type. This method throws a `FormatException` in case the `string` passed to this method can't be converted to a number, and it throws an `OverflowException` if a number can be converted but it doesn't fit into an `int`:

```
void NumberDemo1(string n)
{
  if (n is null) throw new ArgumentNullException(nameof(n));
  try
  {
    int i = int.Parse(n);
    Console.WriteLine($"converted: {i}");
  }
  catch (FormatException ex)
  {
    Console.WriteLine(ex.Message);
  }
  catch (OverflowException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

If the method `NumberDemo1` is typically used only in a way to pass numbers in a string and receiving something other than a number is exceptional, it's okay to program it this way. However, in cases when it's normal for the program flow to expect strings that cannot be converted, you can use the `TryParse` method. This method doesn't throw an exception if the string cannot be converted to a number. Instead, `TryParse` returns `true` if parsing succeeds, and it returns `false` if parsing fails:

```
void NumberDemo2(string n)
{
  if (n is null) throw new ArgumentNullException(nameof(n));
  if (int.TryParse(n, out int result))
  {
    Console.WriteLine($"converted {result}");
  }
  else
  {
    Console.WriteLine("not a number");
  }
}
```

Implementing Multiple Catch Blocks

The easiest way to see how `try...catch...finally` blocks work in practice is with a couple of examples. The first example is called `SimpleExceptions`. It repeatedly asks the user to type a number and then displays it. However, for the sake of this example, imagine that the number must be between 0 and 5; otherwise, the program isn't able to process the number properly. Therefore, you throw an exception if the user types anything outside this range. The program then continues to ask for more numbers for processing until the user presses the Enter key without entering anything.

> **NOTE** *You should note that this code does not provide a good example of when to use exception handling, but it does show good practice on how to use exception handling. As their name suggests, exceptions are provided for other than normal circumstances. Users often type silly things, so this situation doesn't really count. Normally, your program handles incorrect user input by performing an instant check and asking the user to retype the input if it isn't valid. However, generating exceptional situations is difficult in a small example that you can read through in a few minutes, so I'm making do with this less-than-ideal one to demonstrate how exceptions work. The examples that follow present more realistic situations.*

The code for `SimpleExceptions` looks like this (code file `SimpleExceptions/Program.cs`):

```
while (true)
{
  try
  {
    Console.Write("Input a number between 0 and 5 " +
      "(or just hit return to exit)> ");
    string? userInput = Console.ReadLine();

    if (string.IsNullOrEmpty(userInput))
    {
      break;
    }

    int index = Convert.ToInt32(userInput);

    if (index < 0 || index > 5)
    {
      throw new IndexOutOfRangeException($"You typed in {userInput}");
    }
    Console.WriteLine($"Your number was {index}");
  }
  catch (IndexOutOfRangeException ex)
  {
    Console.WriteLine("Exception: " +
      $"Number should be between 0 and 5. {ex.Message}");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"An exception was thrown. Exception type: {ex.GetType().Name} " +
      $"Message: {ex.Message}");
  }
```

```
    finally
    {
      Console.WriteLine("Thank you\n");
    }
  }
```

The core of this code is a `while` loop, which continually uses `ReadLine` to ask for user input. `ReadLine` returns a string, so your first task is to convert it to an `int` using the `System.Convert.ToInt32` method. The `System.Convert` class contains various useful methods to perform data conversions, and it provides an alternative to the `int.Parse` method. In general, `System.Convert` contains methods to perform various type conversions. Recall that the C# compiler resolves `int` to instances of the `System.Int32` base class.

> **NOTE** *It is also worth pointing out that the parameter passed to the* `catch` *block is scoped to that* `catch` *block—which is why you can use the same parameter name,* `ex`*, in successive* `catch` *blocks in the preceding code.*

In the preceding example, you also check for an empty string because it is your condition for exiting the `while` loop. Notice how the `break` statement breaks right out of the enclosing `try` block as well as the `while` loop because this is valid behavior. Of course, when execution breaks out of the `try` block, the `Console.WriteLine` statement in the `finally` block is executed. Although you just display a greeting here, more commonly you will be doing tasks such as closing file handles and calling the `Dispose` method of various objects to perform any cleanup. After the application leaves the `finally` block, it simply carries on executing into the next statement that it would have executed had the `finally` block not been present. In the case of this example, though, you iterate back to the start of the `while` loop and enter the `try` block again (unless the `finally` block was entered as a result of executing the `break` statement in the `while` loop, in which case you simply exit the `while` loop).

Next, you check for your exception condition:

```
if (index < 0 || index > 5)
{
  throw new IndexOutOfRangeException($"You typed in {userInput}");
}
```

When throwing an exception, you need to specify what type of exception to throw. Although the class `System.Exception` is available, it is intended only as a base class. It is considered bad programming practice to throw an instance of this class as an exception because it conveys no information about the nature of the error condition. .NET contains many other exception classes that are derived from `Exception`. Each of these matches a particular type of exception condition, and you are free to define your own as well. The goal is to provide as much information as possible about the particular exception condition by throwing an instance of a class that matches the particular error condition. In the preceding example, `System.IndexOutOfRangeException` is the best choice for the circumstances. `IndexOutOfRangeException` has several constructor overloads. The one chosen in the example takes a string describing the error. Alternatively, you might choose to derive your own custom `Exception` object that describes the error condition in the context of your application.

Suppose that the user next types a number that is not between 0 and 5. The number is picked up by the `if` statement, and an `IndexOutOfRangeException` object is instantiated and thrown. At this point, the application immediately exits the `try` block and hunts for a `catch` block that handles `IndexOutOfRangeException`. The first `catch` block it encounters is this:

```
catch (IndexOutOfRangeException ex)
{
  Console.WriteLine($"Exception: Number should be between 0 and 5." +
    $"{ex.Message}");
}
```

Because this `catch` block takes a parameter of the appropriate class, the `catch` block receives the exception instance and is executed. In this case, you display an error message and the `Exception.Message` property (which corresponds to the string passed to the `IndexOutOfRangeException`'s constructor). After executing this `catch` block, control then switches to the `finally` block, just as if no exception had occurred.

Notice that in the example you have also provided another `catch` block:

```
catch (Exception ex)
{
  Console.WriteLine($"An exception was thrown. Message was: {ex.Message}");
}
```

This `catch` block would also be capable of handling an `IndexOutOfRangeException` if it weren't for the fact that such exceptions will already have been caught by the previous `catch` block. A reference to a base class can also refer to any instances of classes derived from it, and all exceptions are derived from `Exception`. This `catch` block isn't executed because the application executes only the first suitable `catch` block it finds from the list of available `catch` blocks. This catch block isn't executed when an exception of type `IndexOutOfRangeException` is thrown. The application executes only the first suitable catch block it finds from the list of available catch blocks. This second catch block catches other exceptions derived from the `Exception` base class. Be aware that the three separate calls to methods within the `try` block (`Console.ReadLine`, `Console.Write`, and `Convert.ToInt32`) might throw other exceptions.

If the user types something that is not a number—say `a` or `hello`—the `Convert.ToInt32` method throws an exception of the class `System.FormatException` to indicate that the string passed into `ToInt32` is not in a format that can be converted to an `int`. When this happens, the application traces back through the method calls, looking for a handler that can handle this exception. Your first `catch` block (the one that takes an `IndexOutOfRangeException`) will not do. The application then looks at the second `catch` block. This one will do because `FormatException` is derived from `Exception`, so a `FormatException` instance can be passed in as a parameter here.

The structure of the example is fairly typical of a situation with multiple `catch` blocks. You start with `catch` blocks that are designed to trap specific error conditions. Then, you finish with more general blocks that cover any errors for which you have not written specific error handlers. Indeed, the order of the `catch` blocks is important. Had you written the previous two blocks in the opposite order, the code would not have compiled because the second `catch` block is unreachable (the `Exception catch` block would catch all exceptions). Therefore, the uppermost `catch` blocks should be the most granular options available, ending with the most general options.

Now that you have analyzed the code for the example, you can run it. The following output illustrates what happens with different inputs and demonstrates both the `IndexOutOfRangeException` and the `FormatException` being thrown:

```
Input a number between 0 and 5 (or just hit return to exit)> 4
Your number was 4
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 0
Your number was 0
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 10
Exception: Number should be between 0 and 5. You typed in 10
Thank you
Input a number between 0 and 5 (or just hit return to exit)> hello
An exception was thrown. Exception type: FormatException, Message: Input string was not
in a correct format.
Thank you
Input a number between 0 and 5 (or just hit return to exit)>
Thank you
```

Catching Exceptions from Other Code

The previous example demonstrates the handling of two exceptions. One of them, `IndexOutOfRangeException`, was thrown by your own code. The other, `FormatException`, was thrown from inside one of the base classes. It is common for code in a library to throw an exception if it detects that a problem has occurred or if one of the methods has been called inappropriately by being passed the wrong parameters. However, library code rarely attempts to catch exceptions; this is regarded as the responsibility of the client code.

Often, exceptions are thrown from the base class libraries while you are debugging. The process of debugging to some extent involves determining why exceptions have been thrown and removing the causes. Your aim should be to ensure that by the time the code is actually shipped, exceptions occur only in exceptional circumstances and, if possible, are handled appropriately in your code.

System.Exception Properties

The example illustrated the use of only the `Message` property of the exception object. However, a number of other properties are available in `System.Exception`, as shown in the following table:

PROPERTY	DESCRIPTION
`Data`	Enables you to add key/value statements to the exception that can be used to supply extra information about it.
`HelpLink`	A link to a help file that provides more information about the exception.
`HResult`	A numerical value that is assigned to the exception.
`InnerException`	If this exception was thrown inside a `catch` block, then `InnerException` contains the exception object that sent the code into that `catch` block.
`Message`	Text that describes the error condition.
`Source`	The name of the application or object that caused the exception.
`StackTrace`	Provides details about the method calls on the stack (to help track down the method that threw the exception).
`TargetSite`	A .NET reflection object that describes the method that threw the exception.

The property value for `StackTrace` is supplied automatically by the .NET runtime if a stack trace is available. `Source` will always be filled in by the .NET runtime as the name of the assembly in which the exception was raised (though you might want to modify the property in your code to give more specific information), whereas `Data`, `Message`, `HelpLink`, and `InnerException` must be filled in by the code that threw the exception, by setting these properties immediately before throwing the exception. For example, the code to throw an exception might look something like this:

```
if (ErrorCondition)
{
  ClassMyException myException = new("Help!!!!");
  myException.Source = "My Application Name";
  myException.HelpLink = "MyHelpFile.txt";
  myException.Data["ErrorDate"] = DateTime.Now;
  myException.Data.Add("AdditionalInfo", "Contact Bill from the Blue Team");
  throw myException;
}
```

Here, `ClassMyException` is the name of the particular exception class you are throwing. Note that it is common practice for the names of all exception classes to end with `Exception`. The string passed to the constructor sets the `Message` property. In addition, note that the `Data` property is assigned in two possible ways.

Exception Filters

With the many different exception types and the hierarchy of exceptions, the original plan with .NET was as soon as you need to handle errors differently, use a different exception type. With many technologies used with .NET, this turned out to not be a practical scenario. For example, using the Windows Runtime often results in COM exceptions, and you want to handle `COMException` differently based on the error code from this exception. To deal with this, C# 6 was enhanced to support exception filters. A `catch` block runs only if the filter returns `true`. You can have different `catch` blocks that act differently when catching different exception types. In some scenarios, it's useful to have the `catch` blocks act differently based on the content of an exception. When doing network calls, you get a network exception for many different scenarios—for example, if the server is not available or the data supplied do not match the expectations. It's good to react to these errors differently. Some exceptions can be recovered in different ways, whereas with others, the user might need some information.

The following code sample throws the exception of type `MyCustomException` and sets the `ErrorCode` property of this exception (code file `ExceptionFilters/Program.cs`):

```
public static void ThrowWithErrorCode(int code)
{
  throw new MyCustomException("Error in Foo") { ErrorCode = code };
}
```

In the following example, the `try` block safeguards the method invocation with two `catch` blocks. The first catch `block` uses the `when` keyword to filter only exceptions if the `ErrorCode` property equals `405`. The expression for the `when` clause needs to return a Boolean value. If the result is `true`, this `catch` block handles the exception. If it is `false`, other catches are looked for. When passing `405` to the method `ThrowWithErrorCode`, the filter returns `true`, and the first `catch` handles the exception. When passing another value, the filter returns `false`, and the second `catch` handles the exception. With filters, you can have multiple handlers to handle the same exception type.

Of course, you can also remove the second `catch` block and not handle the exception in that circumstance.

```
try
{
  ThrowWithErrorCode(405);
}
catch (MyCustomException ex) when (ex.ErrorCode == 405)
{
  Console.WriteLine($"Exception caught with filter {ex.Message} " +
    $"and {ex.ErrorCode}");
}
catch (MyCustomException ex)
{
  Console.WriteLine($"Exception caught {ex.Message} and {ex.ErrorCode}");
}
```

Rethrowing Exceptions

When you catch exceptions, it's common to rethrow exceptions, which means you can change the exception type while throwing the exception again. With this method, you can give the caller more information about what happened. The original exception might not have enough information about the context of what was going on. You can also log exception information and give the caller different information. For example, for a user running the application, exception information does not really help. A system administrator reading log files can react accordingly.

An issue with rethrowing exceptions is that the caller often needs to find out the reason for what happened with the earlier exception, and where it happened. Depending on how exceptions are thrown, stack trace information might be lost. For you to see the different options on rethrowing exceptions, the sample program `RethrowExceptions` shows the different options.

For this sample, two custom exception types are created. The first one, `MyCustomException`, defines the property `ErrorCode` in addition to the members of the base class `Exception`; the second one, `AnotherCustomException`, supports passing an inner exception (code file `RethrowExceptions/MyCustomException.cs`):

```
public class MyCustomException : Exception
{
  public MyCustomException(string message)
    : base(message) { }

  public int ErrorCode { get; set; }
}

public class AnotherCustomException : Exception
{
  public AnotherCustomException(string message, Exception innerException)
    : base(message, innerException) { }
}
```

The method `HandleAll` invokes the methods `HandleAndThrowAgain`, `HandleAndThrowWithInnerException`, `HandleAndRethrow`, and `HandleWithFilter`. The exception that is thrown is caught to write the exception message as well as the stack trace to the console. To better find what line numbers are referenced from the stack trace, the `#line` preprocessor directive is used that restarts the line numbering. With this, the invocation of the methods using the delegate `m` is in line 114 (code file `RethrowExceptions/Program.cs`):

```
#line 100
public static void HandleAll()
{
  Action[] methods =
  {
    HandleAndThrowAgain,
    HandleAndThrowWithInnerException,
    HandleAndRethrow,
    HandleWithFilter
  };

  foreach (var m in methods)
  {
    try
    {
      m(); // line 114
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
      Console.WriteLine(ex.StackTrace);
      if (ex.InnerException != null)
      {
        Console.WriteLine($"\tInner Exception{ex.InnerException.Message}");
        Console.WriteLine(ex.InnerException.StackTrace);
      }
```

```
        Console.WriteLine();
      }
    }
  }
```

The method `ThrowAnException` is the one to throw the first exception. This exception is thrown in line 8002. During development, it helps to know where this exception is thrown:

```
#line 8000
public static void ThrowAnException(string message)
{
  throw new MyCustomException(message); // line 8002
}
```

Naïvely Rethrowing the Exception

The method `HandleAndThrowAgain` does nothing more than log the exception to the console and throw it again using `throw ex`:

```
#line 4000
public static void HandleAndThrowAgain()
{
  try
  {
    ThrowAnException("test 1");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"Log exception {ex.Message} and throw again");
    throw ex; // you shouldn't do that - line 4009
  }
}
```

After running the application, the following simplified output shows the stack trace (without the namespace and the full path to the code files):

```
Log exception test 1 and throw again
test 1
at Program.HandleAndThrowAgain() in Program.cs:line 4009
at Program.HandleAll() in Program.cs:line 114
```

The stack trace shows the call to the `m` method within the `HandleAll` method, which in turn invokes the `HandleAndThrowAgain` method. The information where the exception is thrown at first is completely lost in the call stack of the final catch. This makes it hard to find the original reason of an error. Usually it's not a good idea to just throw the same exception with `throw` passing the exception object. The C# compiler now gives you the warning `CA2200: re-throwing caught exception changes stack information`.

Changing the Exception

One useful scenario is to change the type of the exception and add information to the error. This is done in the method `HandleAndThrowWithInnerException`. After logging the error, a new exception of type `AnotherCustomException` is thrown to pass `ex` as the inner exception:

```
#line 3000
public static void HandleAndThrowWithInnerException()
{
  try
```

```
    {
      ThrowAnException("test 2"); // line 3004
    }
    catch (Exception ex)
    {
      Console.WriteLine($"Log exception {ex.Message} and throw again");
      throw new AnotherCustomException("throw with inner exception", ex); // 3009
    }
  }
```

By checking the stack trace of the outer exception, you see line numbers 3009 and 114 similar to before. However, the inner exception gives the original reason of the error. It gives the line of the method that invoked the erroneous method (3004) and the line where the original (the inner) exception was thrown (8002):

```
Log exception test 2 and throw again
throw with inner exception
at Program.HandleAndThrowWithInnerException() in Program.cs:line 3009
at Program.HandleAll() in Program.cs:line 114
Inner Exception throw with inner exception
at Program.ThrowAnException(String message) in Program.cs:line 8002
at Program.HandleAndThrowWithInnerException() in Program.cs:line 3004
```

No information is lost this way.

> **NOTE** *When trying to find reasons for an error, take a look at whether an inner exception exists. This often gives helpful information.*

> **NOTE** *When catching exceptions, it's good practice to change the exception when rethrowing. For example, catching an* `SqlException` *can result in throwing a business-related exception such as* `InvalidIsbnException`.

Rethrowing the Exception

If the exception type should not be changed, the same exception can be rethrown just with the `throw` statement. Using `throw` without passing an exception object throws the current exception of the `catch` block and keeps the exception information:

```
#line 2000
public static void HandleAndRethrow()
{
  try
  {
    ThrowAnException("test 3");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"Log exception {ex.Message} and rethrow");
    throw; // line 2009
  }
}
```

With this in place, the stack information is not lost. The exception was originally thrown in line 8002 and rethrown in line 2009. Line 114 contains the delegate `m` that invoked `HandleAndRethrow`:

```
Log exception test 3 and rethrow
test 3
at Program.ThrowAnException(String message) in Program.cs:line 8002
at Program.HandleAndRethrow() in Program.cs:line 2009
at Program.HandleAll() in Program.cs:line 114
```

Using Filters to Add Functionality

When rethrowing exceptions using the `throw` statement, the call stack contains the address of the throw. When you use exception filters, it is possible not to change the call stack at all. Now add a `when` keyword that passes a filter method. This filter method named `Filter` logs the message and always returns `false`. That's why the `catch` block is never invoked:

```
#line 1000
public void HandleWithFilter()
{
  try
  {
    ThrowAnException("test 4"); // line 1004
  }
  catch (Exception ex) when(Filter(ex))
  {
    Console.WriteLine("block never invoked");
  }
}
#line 1500
public bool Filter(Exception ex)
{
  Console.WriteLine($"just log {ex.Message}");
  return false;
}
```

Now when you look at the stack trace, the exception originates in the `HandleAll` method in line 114 that in turn invokes `HandleWithFilter`, line 1004 contains the invocation to `ThrowAnException`, and line 8002 contains the line where the exception was thrown:

```
just log test 4
test 4
at Program.ThrowAnException(String message) in Program.cs:line 8002
at Program.HandleWithFilter() in Program.cs:line 1004
at RethrowExceptions.Program.HandleAll() in Program.cs:line 114
```

> **NOTE** *The primary use of exception filters is to filter exceptions based on a value of the exception. Exception filters can also be used for other effects, such as writing log information without changing the call stack. However, exception filters should be fast running, so you should do only simple checks and avoid side effects. Logging is one of the excusable exceptions.*

What Happens If an Exception Isn't Handled?

Sometimes an exception might be thrown, but there is no `catch` block in your code that is able to handle that kind of exception. The `SimpleExceptions` example can serve to illustrate this. Suppose, for example, that you omitted the `FormatException` and catchall `catch` blocks, and you supplied only the block that traps an `IndexOutOfRangeException`. In that circumstance, what would happen if a `FormatException` were thrown?

The answer is that the .NET runtime would catch it. Later in this section, you learn how you can nest `try` blocks; and, in fact, there is already a nested `try` block behind the scenes in the example. The .NET runtime has effectively placed the entire program inside another huge `try` block—it does this for every .NET program. This `try` block has a `catch` handler that can catch any type of exception. If an exception occurs that your code does not handle, the execution flow simply passes right out of your program and is trapped by this `catch` block in the .NET runtime. However, the results of this probably will not be what you want because the execution of your code is terminated promptly. The user sees a dialog that complains that your code has not handled the exception and provides any details about the exception the .NET runtime was able to retrieve. At least the exception has been caught!

In general, if you are writing an executable, try to catch as many exceptions as you reasonably can and handle them in a sensible way. If you are writing a library, it is normally best to catch exceptions that you can handle in a useful way, or where you can add information to the context and throw other exception types as shown in the previous section. Assume that the calling code handles any errors it encounters.

USER-DEFINED EXCEPTION CLASSES

In the previous section, you already created a user-defined exception. You are now ready to look at a larger example that illustrates exceptions. This example, called `SolicitColdCall`, contains two nested `try` blocks and illustrates the practice of defining your own custom exception classes and throwing another exception from inside a `try` block.

This example assumes that a sales company wants to increase its customer base. The company's sales team is going to phone a list of people to invite them to become customers, a practice known in sales jargon as *cold-calling*. To this end, you have a text file available that contains the names of the people to be cold-called. The file should be in a well-defined format in which the first line contains the number of people in the file and each subsequent line contains the name of the next person. In other words, a correctly formatted file of names might look like this:

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

This version of cold-calling is designed to display the name of the person on the screen (perhaps for the salesperson to read). That is why only the names, and not the phone numbers, of the individuals are contained in the file.

For this example, your program asks the user for the name of the file and then simply reads it in and displays the names of people. That sounds like a simple task, but even so, a couple of things can go wrong and require you to abandon the entire procedure:

- The user might type the name of a file that does not exist. This is caught as a `FileNotFound` exception.
- The file might not be in the correct format. There are two possible problems here. One, the first line of the file might not be an integer. Two, there might not be as many names in the file as the first line of the file indicates. In both cases, you want to trap this oddity as a custom exception that has been written especially for this purpose, `ColdCallFileFormatException`.

There is something else that can go wrong that doesn't cause you to abandon the entire process but does mean you need to abandon a person's name and move on to the next name in the file (and therefore trap it by an inner `try` block). Some people are spies working for rival sales companies, so you obviously do not want to let these people know what you are up to by accidentally phoning one of them. For simplicity, assume that you can identify who the spies are because their names begin with B. Such people should have been screened out when the data file was first prepared, but in case any have slipped through, you need to check each name in the file and throw a `SalesSpyFoundException` if you detect a sales spy. This, of course, is another custom exception object.

Finally, you implement this example by coding a class, `ColdCallFileReader`, which maintains the connection to the cold-call file and retrieves data from it. You code this class in a safe way, which means that its methods all throw exceptions if they are called inappropriately—for example, if a method that reads a file is called before the file has even been opened. For this purpose, you write another exception class: `UnexpectedException`.

Catching the User-Defined Exceptions

Start with the top-level statements of the `SolicitColdCall` sample, which catches your user-defined exceptions. Note that you need to call up file-handling classes in the `System.IO` namespace as well as the `System` namespace (code file `SolicitColdCall/Program.cs`):

```
Console.Write("Please type in the name of the file " +
  "containing the names of the people to be cold called > ");
string? fileName = Console.ReadLine();
if (fileName != null)
{
  ColdCallFileReaderLoop1(fileName);
  Console.WriteLine();
}
Console.ReadLine();

void ColdCallFileReaderLoop1(string fileName)
{
  ColdCallFileReader peopleToRing = new();
  try
  {
    peopleToRing.Open(fileName);
    for (int i = 0; i < peopleToRing.NPeopleToRing; i++)
    {
      peopleToRing.ProcessNextPerson();
    }
    Console.WriteLine("All callers processed correctly");
  }
  catch(FileNotFoundException)
  {
    Console.WriteLine($"The file {fileName} does not exist");
  }
  catch(ColdCallFileFormatException ex)
  {
    Console.WriteLine($"The file {fileName} appears to have been corrupted");
    Console.WriteLine($"Details of problem are: {ex.Message}");
    if (ex.InnerException != null)
    {
      Console.WriteLine($"Inner exception was: {ex.InnerException.Message}");
    }
  }
```

```
    catch(Exception ex)
    {
      Console.WriteLine($"Exception occurred:\n{ex.Message}");
    }
    finally
    {
      peopleToRing.Dispose();
    }
  }
```

This code is a little more than just a loop to process people from the file. You start by asking the user for the name of the file. Then you instantiate an object of a class called `ColdCallFileReader`, which is defined shortly. The `ColdCallFileReader` class is the class that handles the file reading. Notice that you do this outside the initial `try` block—that's because the variables that you instantiate here need to be available in the subsequent `catch` and `finally` blocks, and if you declare them inside the `try` block, they would go out of scope at the closing curly brace of the `try` block, where the compiler would complain about it.

In the `try` block, you open the file (using the `ColdCallFileReader.Open` method) and loop over all the people in it. The `ColdCallFileReader.ProcessNextPerson` method reads in and displays the name of the next person in the file, and the `ColdCallFileReader.NPeopleToRing` property indicates how many people should be in the file (obtained by reading the file's first line). There are three `catch` blocks: one for `FileNotFoundException`, one for `ColdCallFileFormatException`, and one to trap any other .NET exceptions.

> **NOTE** *With the sample application, the object of type* `ColdCallFileReader` *is instantiated outside of the* `try` *block. It's a good practice to create constructors that can't fail and don't take a long processing time. In case you use such types, you can create an outer* `try/catch` *block or declare the variable outside the* `try` *block and instantiate it within the* `try` *block.*

In the case of a `FileNotFoundException`, you display a message to that effect. Notice that in this `catch` block, the exception instance is not actually used at all. This `catch` block is used to illustrate the user-friendliness of the application. Exception objects generally contain technical information that is useful for developers, but not the sort of stuff you want to show to end users. Therefore, in this case, you create a simpler message of your own.

For the `ColdCallFileFormatException` handler, you have done the opposite, specifying how to obtain fuller technical information, including details about the inner exception, if one is present.

Finally, if you catch any other generic exceptions, you display a user-friendly message, instead of letting any such exceptions fall through to the .NET runtime.

The `finally` block is there to clean up resources. In this case, that means closing any open file—performed by the `ColdCallFileReader.Dispose` method.

> **NOTE** *C# offers the* `using` *statement and the* `using` *declaration where the compiler itself creates a* `try/finally` *block calling the* `Dispose` *method in the* `finally` *block. The* `using` *keyword can be used with objects implementing the* `IDisposable` *interface. You can read the details of the* `using` *statement and declaration in Chapter 13, "Managed and Unmanaged Memory."*

Throwing the User-Defined Exceptions

Now take a look at the definition of the class that handles the file reading and (potentially) throws your user-defined exceptions: `ColdCallFileReader`. Because this class maintains an external file connection, you need to ensure that it is disposed of correctly in accordance with the principles outlined for the disposing of objects in Chapter 13. Therefore, you derive this class from `IDisposable`.

First, you declare some private fields (code file `SolicitColdCall/ColdCallFileReader.cs`):

```
public class ColdCallFileReader: IDisposable
{
  private FileStream? _fileStream;
  private StreamReader? _streamReader;
  private uint _nPeopleToRing;
  private bool _isDisposed = false;
  private bool _isOpen = false;
```

`FileStream` and `StreamReader`, both in the `System.IO` namespace, are the base classes that you use to read the file. `FileStream` enables you to connect to the file in the first place, whereas `StreamReader` is designed to read text files and implements a method, `ReadLine`, which reads a line of text from a file. You look at `StreamReader` more closely in Chapter 18, "Files and Streams," which discusses file handling in depth.

The `_isDisposed` field indicates whether the `Dispose` method has been called. `ColdCallFileReader` is implemented so that after `Dispose` has been called, it is not permitted to reopen connections and reuse the object. `_isOpen` is also used for error checking—in this case, checking whether the `StreamReader` actually connects to an open file.

The process of opening the file and reading in that first line—the one that tells you how many people are in the file—is handled by the `Open` method:

```
public void Open(string fileName)
{
  if (_isDisposed)
  {
    throw new ObjectDisposedException(nameof(ColdCallFileReader));
  }

  _fileStream = new(fileName, FileMode.Open);
  _streamReader = new(_fileStream);

  try
  {
    string? firstLine = _streamReader.ReadLine();
    if (firstLine != null)
    {
      _nPeopleToRing = uint.Parse(firstLine);
      _isOpen = true;
    }
  }
  catch (FormatException ex)
  {
    throw new ColdCallFileFormatException(
      $"First line isn't an integer {ex}");
  }
}
```

The first thing you do in this method (as with all other `ColdCallFileReader` methods) is check whether the client code has inappropriately called it after the object has been disposed of and, if so, throw a predefined `ObjectDisposedException` object. The `Open` method checks the `_isDisposed` field to determine whether `Dispose` has already been called. Because calling `Dispose` implies that the caller has now finished with this object, you regard it as an error to attempt to open a new file connection if `Dispose` has been called.

Next, the method contains a `try`/`catch` block. The purpose of this one is to catch any errors resulting from a file in which the first line does not contain an integer. If the method `uint.Parse` cannot parse the first line successfully, a `FormatException` can be thrown. If that problem arises, the exception is caught and converted to a more meaningful exception that indicates a problem with the format of the cold-call file. Note that `System.FormatException` is there to indicate format problems with basic data types, not with files, so it's not a particularly useful exception to pass back to the calling routine in this case. The new exception thrown will be trapped by the outermost `try` block. Because no cleanup is needed here, there is no need for a `finally` block. Other exceptions that can happen such as the `IOException` on calling the `ReadLine` method are not caught here and are forwarded to the next `try` block.

If everything is fine, you set the `_isOpen` field to `true` to indicate that there is now a valid file connection from which data can be read.

The `ProcessNextPerson` method also contains an inner `try` block:

```
public void ProcessNextPerson()
{
  if (_isDisposed)
  {
    throw new ObjectDisposedException(nameof(ColdCallFileReader));
  }

  if (!_isOpen)
  {
    throw new UnexpectedException(
      "Attempted to access coldcall file that is not open");
  }

  try
  {
    string? name = _streamReader?.ReadLine();
    if (name is null)
    {
      throw new ColdCallFileFormatException("Not enough names");
    }
    if (name[0] is 'B')
    {
      throw new SalesSpyFoundException(name);
    }
    Console.WriteLine(name);
  }
  catch(SalesSpyFoundException ex)
  {
    Console.WriteLine(ex.Message);
  }
  finally
  {
  }
}
```

Two possible problems can exist when reading the file in the `ProcessNextPerson` method (assuming there actually is an open file connection that is checked first). The first error that is handled is when `null` is returned from the `ReadLine` method. This method returns `null` if it has gone past the end of the file. Because the file contains the number of names at the start of the file, a `ColdCallFileFormatException` is thrown on a mismatch, and there are fewer names than there should be. This is then caught by the outer exception handler, which causes the execution to terminate.

With the second, the line is accessed. If it is discovered that the name is a sales spy, a `SalesSpyFoundException` is thrown. Because that exception has been caught here, inside the loop, it means that execution can subsequently continue in the `Main` method of the program, and the subsequent names in the file continue to be processed.

Again, you don't need a `finally` block here because there is no cleanup to do; however, this time an empty `finally` block is included just to show that you can do so, if you want.

The example is nearly finished. You have just two more members of `ColdCallFileReader` to look at: the `NPeopleToRing` property, which returns the number of people who are supposed to be in the file, and the `Dispose` method, which closes an open file. Notice that the `Dispose` method returns immediately if it has already been called—this is the recommended way of implementing it. It also confirms that there actually is a file stream to close before closing it. This example is shown here to illustrate defensive coding techniques:

```
public uint NPeopleToRing
{
  get
  {
    if (_isDisposed)
    {
      throw new ObjectDisposedException("peopleToRing");
    }
    if (!_isOpen)
    {
      throw new UnexpectedException(
        "Attempted to access cold-call file that is not open");
    }
    return _nPeopleToRing;
  }
}

public void Dispose()
{
  if (_isDisposed)
  {
    return;
  }
  _isDisposed = true;
  _isOpen = false;

  _streamReader?.Dispose();
  _streamReader = null;
}
```

Defining the User-Defined Exception Classes

Finally, you need to define three of your own exception classes. Defining your own exception is quite easy because there are rarely any extra methods to add. It is just a case of implementing a constructor to ensure that the base class constructor is called correctly. Here is the full implementation of `SalesSpyFoundException` (code file `SolicitColdCall/SalesSpyFoundException.cs`):

```
public class SalesSpyFoundException: Exception
{
  public SalesSpyFoundException(string spyName)
    : base($"Sales spy found, with name {spyName}") { }

  public SalesSpyFoundException(string spyName, Exception innerException)
    : base($"Sales spy found with name {spyName}", innerException) {  }
}
```

Notice that it is derived from `Exception`, as you would expect for a custom exception. In fact, in practice, you would probably have added an intermediate class, something like `ColdCallFileException`, derived from `Exception`, and then derived both of your exception classes from this class. This ensures that the handling code has that extra-fine degree of control over which exception handler handles each exception. However, to keep the example simple, you will not do that.

You have done one bit of processing in `SalesSpyFoundException`. You have assumed that the message passed into its constructor is just the name of the spy found, so you turn this string into a more meaningful error message. You have also provided two constructors: one that simply takes a message, and one that also takes an inner exception as a parameter. When defining your own exception classes, it is best to include, at a minimum, at least these two constructors (although you will not actually be using the second `SalesSpyFoundException` constructor in this example).

The `ColdCallFileFormatException` follows the same principles as the previous exception, but you don't do any processing on the message (code file `SolicitColdCall/ColdCallFileFormatException.cs`):

```
public class ColdCallFileFormatException: Exception
{
  public ColdCallFileFormatException(string message)
    : base(message) {}

  public ColdCallFileFormatException(string message, Exception innerException)
    : base(message, innerException) {}
}
```

Finally, you have `UnexpectedException`, which looks much the same as `ColdCallFileFormatException` (code file `SolicitColdCall/UnexpectedException.cs`):

```
public class UnexpectedException: Exception
{
  public UnexpectedException(string message)
    : base(message) { }

  public UnexpectedException(string message, Exception innerException)
    : base(message, innerException) { }
}
```

Now you are ready to test the program. First, try the `people.txt` file. The contents are defined here:

```
4
George Washington
Benedict Arnold
```

```
John Adams
Thomas Jefferson
```

This has four names (which match the number given in the first line of the file), including one spy. Then try the following `people2.txt` file, which has an obvious formatting error:

```
49
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

Finally, try the example, but specify the name of a file that does not exist, such as `people3.txt`. Running the program three times for the three filenames returns these results:

```
SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
All callers processed correctly

SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people2.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
The file people2.txt appears to have been corrupted.
Details of the problem are: Not enough names

SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people3.txt
The file people3.txt does not exist.
```

This application has demonstrated a number of different ways in which you can handle the errors and exceptions that you might find in your own applications.

CALLER INFORMATION

When dealing with errors, it is often helpful to get information about the error where it occurred. Earlier in this chapter, the `#line` preprocessor directive was used to change the line numbering of the code to get better information with the call stack. A method can get caller information through optional parameters. You can use attributes to get the line numbers, filenames, and member names from within code. The attributes `CallerLineNumber`, `CallerFilePath`, and `CallerMemberName` defined within the namespace `System.Runtime.CompilerServices` are directly supported by the C# compiler, which sets these values.

The `Log` method from the following code snippet demonstrates how to use these attributes. With the implementation, the information is written to the console (code file `CallerInformation/Program.cs`):

```
public void Log([CallerLineNumber] int line = -1,
  [CallerFilePath] string path = default,
  [CallerMemberName] string name = default)
```

```
{
  Console.WriteLine($"Line {line}");
  Console.WriteLine(path);
  Console.WriteLine(name);
  Console.WriteLine();
}
```

Let's invoke this method with some different scenarios. In the following `Main` method, the `Log` method is called by using an instance of the `Program` class, within the set accessor of the property, and within a lambda expression. Argument values are not assigned to the method, enabling the compiler to fill them in:

```
public static void Main()
{
  Program p = new();
  p.Log();
  p.SomeProperty = 33;
  Action a1 = () => p.Log();
  a1();
}

private int _someProperty;
public int SomeProperty
{
  get => _someProperty;
  set
  {
    Log();
    _someProperty = value;
  }
}
```

The result of the running program is shown next. Where the `Log` method was invoked, you can see the line numbers, the filename, and the caller member name. With the `Log` inside the `Main` method, the member name is `Main`. The invocation of the `Log` method inside the set accessor of the property `SomeProperty` shows `SomeProperty`. The `Log` method inside the lambda expression doesn't show the name of the generated method, but instead the name of the method where the lambda expression was invoked (`Main`), which is more useful, of course.

```
Line 9
C:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
Main

Line 21
C:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
SomeProperty

Line 11
C:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
Main
```

Using the `Log` method within a constructor, the caller member name shows `ctor`. With a destructor, the caller member name is `Finalize`, as this is the method name generated.

> **NOTE** *The destructor and finalizer are covered in Chapter 13.*

> **NOTE** *A great use of the* `CallerMemberName` *attribute is with the implementation of the interface* `INotifyPropertyChanged`*. This interface requires the name of the property to be passed with the method implementation. You can see the implementation of this interface in Chapter 30, "Patterns with XAML Apps."*

SUMMARY

This chapter examined the rich mechanism C# provides for dealing with error conditions through exceptions. You are not limited to the generic error codes that could be output from your code; instead, you have the capability to go in and uniquely handle the most granular of error conditions. Sometimes these error conditions are provided to you through .NET itself; at other times, though, you might want to code your own error conditions as illustrated in this chapter. In either case, you have many ways to protect the workflow of your applications from unnecessary and dangerous faults.

Detailed information on logging errors is covered in Chapter 16, "Diagnostics and Metrics."

The next chapter goes into important keywords for asynchronous programming: `async` and `await`.